C# Codes for Design Pattern Implementations

Singleton Design Pattern

1. Early Instantiation

Instance is created at the loading time.

Code1

```
public class EarlySingleton
{
   // Early instantiation with a static readonly field
   private static readonly EarlySingleton _instance = new EarlySingleton();

   // Private constructor to prevent direct instantiation
   private EarlySingleton()
   {
      // Initialization code, if any
   }

   // Public static method to access the singleton instance
   public static EarlySingleton Instance
   {
      get { return _instance; }
   }

   // Other methods and properties of the singleton class can be added as needed
}
```

Code usage 1

```
class Program
{
   static void Main()
   {
      // Accessing the singleton instance
      EarlySingleton instance = EarlySingleton.Instance;

      // You can use 'instance' to access methods and properties of the singleton class
   }
}
```

2. Lazy Instantiation

Instance is created when required.

-code in slides-

Factory Design Pattern

code

```
using System;

// Product
public abstract class Product
{
   public abstract string Operation();
}

// ConcreteProduct
public class ConcreteProductA : Product
{
```

```
    public override string Operation()
    {
        return "ConcreteProductA operation";
    }
}

// Creator
public abstract class Creator
{
    public abstract Product FactoryMethod();

    public string SomeOperation()
    {
        Product product = FactoryMethod();
        return $"Creator: {product.Operation()}";
    }
}

// ConcreteCreator
public class ConcreteCreatorA : Creator
{
    public override Product FactoryMethod()
    {
        return new ConcreteProductA();
    }
}

// Client code
class Program
```

```
{
    static void Main()
    {
        ClientCode(new ConcreteCreatorA());
    }

    public static void ClientCode(Creator creator)
    {
        Console.WriteLine(creator.SomeOperation());
    }
}
```

description

- Product is an abstract class with an abstract method Operation.
- ConcreteProductA is a concrete class that extends Product and implements the Operation method.
- Creator is an abstract class with an abstract method FactoryMethod and a method SomeOperation that uses the factory method to create a Product.
- ConcreteCreatorA is a concrete class that extends Creator and implements the FactoryMethod to create a ConcreteProductA.
- The client code (Main method) uses ConcreteCreatorA to create a product without knowing the specific class of the product.

FACADE design pattern

Code

```
// Subsystem components
class CPU
{
    public void Start()
    {
```

```
        Console.WriteLine("CPU started");
    }

    public void Execute()
    {
        Console.WriteLine("CPU executing");
    }
}

class Memory
{
    public void Load()
    {
        Console.WriteLine("Memory loaded");
    }
}

class HardDrive
{
    public void Read()
    {
        Console.WriteLine("HardDrive reading");
    }
}

// Facade
class ComputerFacade
{
```

```
    private CPU cpu;
    private Memory memory;
    private HardDrive hardDrive;

    public ComputerFacade()
    {
        this.cpu = new CPU();
        this.memory = new Memory();
        this.hardDrive = new HardDrive();
    }

    public void StartComputer()
    {
        cpu.Start();
        memory.Load();
        hardDrive.Read();
        cpu.Execute();
    }
}

// Client code
class Client
{
    static void Main(string[] args)
    {
        ComputerFacade computer = new ComputerFacade();
        computer.StartComputer();
    }
```

```
}
```

description

Demonstrating the Facade pattern with a ComputerFacade coordinating the actions of subsystem components (CPU, Memory, HardDrive). The client code only interacts with the ComputerFacade, providing a simplified interface to start the computer.

Adaptor design pattern

Code

```
using System;


// Target interface
interface IPrinter
{
    void Print();
}


// Adaptee class with an incompatible interface
class LegacyPrinter
{
    public void PrintWithLegacyFormat()
    {
        Console.WriteLine("Printing using legacy format.");
    }
}


// Adapter class that implements the target interface and adapts the legacy printer
class PrinterAdapter : IPrinter
```

```
{
    private readonly LegacyPrinter legacyPrinter;

    public PrinterAdapter(LegacyPrinter legacyPrinter)
    {
        this.legacyPrinter = legacyPrinter;
    }

    public void Print()
    {
        // Adapting the legacy method to fit the new interface
        legacyPrinter.PrintWithLegacyFormat();
    }
}

// Client code that expects the IPrinter interface
class Client
{
    public void PrintUsingPrinter(IPrinter printer)
    {
        printer.Print();
    }
}

class Program
{
    static void Main()
    {
```

```
        // Using the legacy printer with the adapter
        LegacyPrinter legacyPrinter = new LegacyPrinter();
        PrinterAdapter adapter = new PrinterAdapter(legacyPrinter);

        // Using the client code with the adapted printer
        Client client = new Client();
        client.PrintUsingPrinter(adapter);
    }
}
```

Description

In this example, IPrinter is the target interface that the client code expects, LegacyPrinter is the existing class with an incompatible interface, and PrinterAdapter is the adapter class that bridges the gap between them. The PrinterAdapter implements the IPrinter interface and uses an instance of LegacyPrinter to adapt the interface and make it compatible with the client's expectations.

Decorator design pattern

Code

```
using System;

// Component interface
public interface ICoffee
{
    int Cost();
}

// Concrete Component
public class SimpleCoffee : ICoffee
{
```

```
    public int Cost()
    {
        return 5;
    }
}

// Decorator
public abstract class CoffeeDecorator : ICoffee
{
    private readonly ICoffee _coffee;

    public CoffeeDecorator(ICoffee coffee)
    {
        _coffee = coffee ?? throw new ArgumentNullException(nameof(coffee));
    }

    public virtual int Cost()
    {
        return _coffee.Cost();
    }
}

// Concrete Decorator
public class MilkDecorator : CoffeeDecorator
{
    public MilkDecorator(ICoffee coffee) : base(coffee)
    {
    }
```

```
    public override int Cost()
    {
        return base.Cost() + 2;
    }
}

// Concrete Decorator
public class SugarDecorator : CoffeeDecorator
{
    public SugarDecorator(ICoffee coffee) : base(coffee)
    {
    }

    public override int Cost()
    {
        return base.Cost() + 1;
    }
}

class Program
{
    static void Main()
    {
        // Client code
        ICoffee simpleCoffee = new SimpleCoffee();
        Console.WriteLine("Cost of simple coffee: " + simpleCoffee.Cost());
```

```
        ICoffee milkCoffee = new MilkDecorator(simpleCoffee);
        Console.WriteLine("Cost of milk coffee: " + milkCoffee.Cost());

        ICoffee sugarMilkCoffee = new SugarDecorator(milkCoffee);
        Console.WriteLine("Cost of sugar milk coffee: " + sugarMilkCoffee.Cost());
    }
}
```

Description

SimpleCoffee is the base component, MilkDecorator and SugarDecorator are decorators, and the client can combine these decorators to create customized coffee objects with added functionalities.

Template design pattern

Code

```
using System;

// Abstract class defining the template method
abstract class HotBeverage
{
    public void PrepareBeverage()
    {
        BoilWater();
        Brew();
        PourInCup();
        AddCondiments();
    }
```

```
    protected void BoilWater()
    {
        Console.WriteLine("Boiling water");
    }

    protected abstract void Brew();

    protected void PourInCup()
    {
        Console.WriteLine("Pouring into cup");
    }

    protected abstract void AddCondiments();
}

// Concrete class for making tea
class Tea : HotBeverage
{
    protected override void Brew()
    {
        Console.WriteLine("Steeping the tea");
    }

    protected override void AddCondiments()
    {
        Console.WriteLine("Adding lemon");
    }
}
```

```
// Concrete class for making coffee
class Coffee : HotBeverage
{
    protected override void Brew()
    {
        Console.WriteLine("Dripping coffee through filter");
    }

    protected override void AddCondiments()
    {
        Console.WriteLine("Adding sugar and milk");
    }
}

class Program
{
    static void Main()
    {
        // Making tea
        HotBeverage tea = new Tea();
        Console.WriteLine("Making tea:");
        tea.PrepareBeverage();

        Console.WriteLine();

        // Making coffee
        HotBeverage coffee = new Coffee();
```

```
        Console.WriteLine("Making coffee:");
        coffee.PrepareBeverage();
    }
}
```

Description

HotBeverage is an abstract class representing the template for making hot beverages. It defines the steps of the process (BoilWater, Brew, PourInCup, AddCondiments) using the template method PrepareBeverage.

Tea and Coffee are concrete classes that extend HotBeverage and provide specific implementations for the abstract methods (Brew and AddCondiments).

The Main method demonstrates creating instances of Tea and Coffee and calling the PrepareBeverage method to make each type of beverage.

Chain of Responsibility Design Pattern

code

```
using System;


// Handler interface
public abstract class Handler
{
    protected Handler NextHandler;


    public void SetNextHandler(Handler handler)
    {
        NextHandler = handler;
    }


    public abstract void HandleRequest(string request);
}
```

```
// Concrete Handlers
public class ConcreteHandlerA : Handler
{
    public override void HandleRequest(string request)
    {
        if (request == "A")
        {
            Console.WriteLine("ConcreteHandlerA handles the request.");
        }
        else if (NextHandler != null)
        {
            NextHandler.HandleRequest(request);
        }
    }
}

public class ConcreteHandlerB : Handler
{
    public override void HandleRequest(string request)
    {
        if (request == "B")
        {
            Console.WriteLine("ConcreteHandlerB handles the request.");
        }
        else if (NextHandler != null)
        {
            NextHandler.HandleRequest(request);
```

```
        }
    }
}

// Client
public class Client
{
    private Handler handlerChain;

    public Client()
    {
        handlerChain = new ConcreteHandlerA();
        handlerChain.SetNextHandler(new ConcreteHandlerB());
    }

    public void MakeRequest(string request)
    {
        handlerChain.HandleRequest(request);
    }
}

// Usage
class Program
{
    static void Main()
    {
        Client client = new Client();
        client.MakeRequest("A");  // ConcreteHandlerA handles the request.
```

```
        client.MakeRequest("B");  // ConcreteHandlerB handles the request.
        client.MakeRequest("C");  // No handler can handle the request.
    }
}
```

Description

Handler is an abstract class representing the handler interface, and ConcreteHandlerA and ConcreteHandlerB are concrete handlers. The Client initiates the requests and the handlers are linked together to form a chain. Each handler decides whether to process the request or pass it to the next handler in the chain.

Proxy design pattern

Code

```
using System;


// Subject interface
interface Image
{
    void Display();
}


// RealSubject
class RealImage : Image
{
    private string filename;


    public RealImage(string filename)
    {
```

```
        this.filename = filename;
        LoadImageFromDisk();
    }

    private void LoadImageFromDisk()
    {
        Console.WriteLine($"Loading image: {filename}");
    }

    public void Display()
    {
        Console.WriteLine($"Displaying image: {filename}");
    }
}

// Proxy
class ProxyImage : Image
{
    private RealImage realImage;
    private string filename;

    public ProxyImage(string filename)
    {
        this.filename = filename;
    }

    public void Display()
    {
```

```
        if (realImage == null)
        {
            realImage = new RealImage(filename);
        }
        realImage.Display();
    }
}

// Client code
class ProxyPatternExample
{
    static void Main()
    {
        Image image1 = new ProxyImage("cat.jpg");
        Image image2 = new ProxyImage("dog.jpg");

        // The real image is only loaded and displayed when necessary
        image1.Display(); // Loading image: cat.jpg, Displaying image: cat.jpg
        image1.Display(); // Displaying image: cat.jpg (no reloading)

        image2.Display(); // Loading image: dog.jpg, Displaying image: dog.jpg
    }
}
```

Description

Demonstrating the Proxy Design Pattern with a simple image loading scenario. The Image interface defines the common methods, RealImage implements the real object, and ProxyImage acts as the proxy controlling access to the real object. The client code creates instances of the proxy and uses them to interact with the real object as needed.